U.S. PATENT APPLICATION

Title: Synchronizing Use of a Device By Multiple Software Components in Accordance with Information Stored At The Device

Inventor(s): Avigdor Eldar

Filing Date: September 12, 2003

Docket No.: P16496

Prepared by: Patrick Buckley
Buckley, Maschoff, Talwalkar & Allison LLC
Five Elm Street
New Canaan, CT 06840
(203) 972-0191

SYNCHRONIZING USE OF A DEVICE BY MULTIPLE SOFTWARE COMPONENTS IN ACCORDANCE WITH INFORMATION STORED AT THE DEVICE

BACKGROUND

Although multiple software components might be able to use a single device, in some cases the device should only be controlled by one software component at any given time. For example, in some circumstances a first software component might access a network processor to exchange information using a secure network protocol while at other times a second component uses the network processor as a general encryption and/or decryption engine. As a result, before using a device a software component might attempt to make sure that no other software component is currently using that device.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a system.

FIG. 2 is a flow chart of a method according to some embodiments.

FIG. 3 is a block diagram of a system according to some embodiments.

FIG. 4 is a flow chart of an initialization method according to some embodiments.

FIG. 5 is a flow chart of a software component method according to some embodiments.

FIG. 6 is a block diagram of a system according to one embodiment.

DETAILED DESCRIPTION

Multiple software components may use a single device. For example, FIG. 1 is a block diagram of a system 100 that includes an operating system 120, such as an operating system executing at an INTEL® PENTIUM® 4 processor. A first and second

software component residing in the operating system 120 logic space can each use a single device 110. Moreover, the device 110 should only be used by one of the software components at any given time. Thus, before using the device 110 the first software component might attempt to make sure that the second software component is not currently using the device 110.

To facilitate this process, a "synchronization object" or "semaphore" might be used. For example, the two software components might use a shared host memory location accessible by the processor 100 to store an indication of whether or not the device is currently being used.

In some cases, however, the software components might not be able to exchange information using host memory (*e.g.*, independent first and second software components might be unable to access a shared host memory location). For example, a Linux kernel or software components might reside in different virtual memory address regions (*e.g.*, one in kernel space and the other in user space).

FIG. 2 is a flow chart of a method according to some embodiments. The flow charts described herein do not necessarily imply a fixed order to the actions, and embodiments may be performed in any order that is practicable. The method of FIG. 2 may be associated with, for example, the first software component described with respect to FIG. 1. Note that any of the methods described herein may be performed by hardware, software (including microcode), or a combination of hardware and software. For example, a storage medium may store thereon instructions that when executed by a machine result in performance according to any of the embodiments described herein.

At 202, it is determined that a first software component is to use a device, the device being shared with a second software component. At 204, it is arranged via information stored at the device to use the device when the second software component is not using the device. That is, by storing a synchronization object, a semaphore, or other information to (and retrieving information from) the device before using the device, the

first software component may check to make sure that no other software component is currently using the device.

A number of different techniques could be used to synchronize access to the device, and FIGS. 3 through 5 illustrate one such technique (referred to as "Peterson's Algorithm"). In particular, FIG. 3 is a block diagram of a system 300 in which two software components associated with an operating system 320 might need to use a single device 310 according to some embodiments. The software components may, for example, access the device 310 through an interface that operates in accordance with the Peripheral Component Interconnect (PCI) Standards Industry Group (SIG) standard entitled "Conventional PCI 2.2" or "PCI Express 1.0."

The device 310 could be for example, a network adapter that supports encryption and decryption acceleration in order to exchange IPsec information as defined by the Internet Engineering Task Force (IETF) Request For Comment (RFC) number 2401 entitled "Security Architecture for the Internet Protocol" (November 1998). In this case, the first software component might be a network driver and the second software component might be an encryption driver (*e.g.*, to let the operating system 320 take advantage of the encryption and decryption acceleration).

As another example, the device 310 might be a network adapter that supports remote access, such as an adapter configured in accordance with the Distributed Management Task Force (DMTF) specification entitled "Alert Standard Format (ASF) Version 2.0" (April 2003). In this case, the first software component might be a network driver and the second software component might be a different driver that enables the adapter to act as an ASF alerting device (*e.g.*, because different functionality might be exposed by each driver).

As still another example, the device 310 might be a PCI device that exposes an additional interface to a host bus that operates in accordance with the Smart Battery System (SBS) Implementer's Forum specification entitled "System Management Bus

(SMBus) Version 2.0" (August 2000). In this case, one of the software components might be a driver associated with an SMBus controller.

As yet another example, the first software component might be a run-time driver for a peripheral device while the second software component is a diagnostic driver for the device (*e.g.*, the run-time driver being used to exchange information with a PCI disk controller and the diagnostic driver being used to test the controller).

In accordance with Peterson's Algorithm (provided herein as only one example of a synchronization technique), three items of information may be stored at the device 310. In particular, a first component flag and a second component flag may each indicate either "free" or "busy." In addition, a turn flag may indicate either "first component" or "second component." The three items of information might be stored, for example, in registers at the device. According to some embodiments, the registers are "pre-fetchable" volatile memory (*e.g.*, the software components may exchange information with the registers without adversely altering operation of the device 310).

FIG. 4 is a flow chart of an initialization method according to some embodiments. The method may be performed, for example, by the operating system 320 or the device 310. At 402, the first component flag is set to "free, and the second component flag is set to "free" at 404. At 406, the turn flag is set to "first component." According to other embodiments, the turn flag may instead be set to either "first component" or "second component" (*e.g.*, on a random basis). The following is one pseudo-code representation of an initialization process that may be performed in connection with the Peterson's Algorithm:

```
flag: array[0..1] of Boolean;
turn:  0..1;
flag[0] = FALSE;
flag[1] = FALSE;
turn := random (0..1)
```

where flag[0] and flag [1] are the first and second component flags, respectively, and either may have a value of "FALSE" (indicating "free") or "TRUE" (indicating "busy").

Moreover, turn is the turn flag and may have a value of "0" (indicating "first component") or "1" (indicating "second component").

FIG. 5 is a flow chart of a software component method according to some embodiments. The method may be performed, for example, by the first software component described with respect to FIG. 3. At 502, it is determined that a device is to be used. For example, the first software component may determine that it needs to access a network adapter that can also be accessed by the second software component.

At 504, the first component flag is set to "busy," and the turn flag is set to "second component" at 506. The first software component then waits until either the second component flag indicates "free" (at 508) or the turn flag indicates "first component" (at 512). When either of these two conditions is met, the first software component may use the device at 510. After use of the device is completed, the first software component sets the first component flag to "free" (letting the second software component use the device if needed).

A method similar to that described with respect to FIG. 5 may be provided with respect to the second software component. The following is one pseudo-code representation of a software component process that may be performed in connection with the Peterson's Algorithm:

```
entry protocol for software component i

repeat {
flag[i] := TRUE;
turn := j;
while (flag[j] and turn = j) do { };

[software component i code using the device]

exit protocol for software component i

flag[i] := false;
```

where $i = 0$ and $j = 1$ when the code is executed by the first software component and $i = 1$ and $j = 0$ when the code is executed by the second software component. Note that

accesses to the device 310 by the operating system 320 might comprise atomic operations (*e.g.*, a PCI bus would make sure that two separate accesses are serialized).

FIG. 6 is a block diagram of a system 600 according to one embodiment. The system 600 may comprise, for example, a Personal Computer (PC), a server, a workstation, or a mobile processing device. The system 600 includes a network adapter 610 to exchange packets of information via a port 612 in accordance with the Fast Ethernet LAN transmission standard 802.3-2002® published by the Institute of Electrical and Electronics Engineers (IEEE). One example of such a network adapter 610 is the INTEL® PRO/1000 Gigabit Server Adapter. Moreover, the system 600 includes an operating system 620, and either a network driver or an encryption driver may use the network adapter 610 in accordance with any of the embodiments described herein. For example, information stored at the network adapter 610 could be used to make sure that the encryption driver does not access the network adapter 610 when it is currently being accessed by the network driver.

The following illustrates various additional embodiments. These do not constitute a definition of all possible embodiments, and those skilled in the art will understand that many other embodiments are possible. Further, although the following embodiments are briefly described for clarity, those skilled in the art will understand how to make any changes, if necessary, to the above description to accommodate these and other embodiments and applications.

For example, although some embodiments have been described with respect to the synchronization of two software components, embodiments may be used with respect to more than two software components (*e.g.*, by using an additional register to store a third component flag and increasing the size of the turn flag register so that it able to store three different values).

In addition, although specific examples of devices and software components have been provided, embodiments may be used with respect to other types of devices and/or software components (*e.g.*, associated with concurrent sequential software processes that

compete for a finite resource). Similarly, embodiments are not limited to PCI devices. For example, synchronization may be provided for software components that access a device via an interface in accordance with the "Universal Serial Bus (USB) Specification Revision 2.0" (April 2000) available from the USB Implementers Forum.

Moreover, although some embodiments have been described with respect to Peterson's Algorithm, embodiments may use any other synchronization technique. Note that some synchronization techniques may provide determinism (*e.g.*, the outputs are determined by the inputs), mutual exclusion (*e.g.*, two software components will not access a device at the same time), progress (*e.g.*, no software component is blocked except by another that is using, or waiting to use, a device), deadlock avoidance, bounded waiting (*e.g.*, a theoretical limit on how long a software component may need to wait before accessing a device), and/or varying degrees of performance and fairness. Embodiments of the present invention, however, may include any combination of these characteristics (*e.g.*, some embodiments might not provide completely bounded waiting). By way of example, Dekker's Algorithm could be used to synchronize use of a device by software components.

The several embodiments described herein are solely for the purpose of illustration. Persons skilled in the art will recognize from this description other embodiments may be practiced with modifications and alterations limited only by the claims.